

OpenACC cache Directive: Opportunities and Optimizations

Ahmad Lashgar & Amirali Baniyasadi

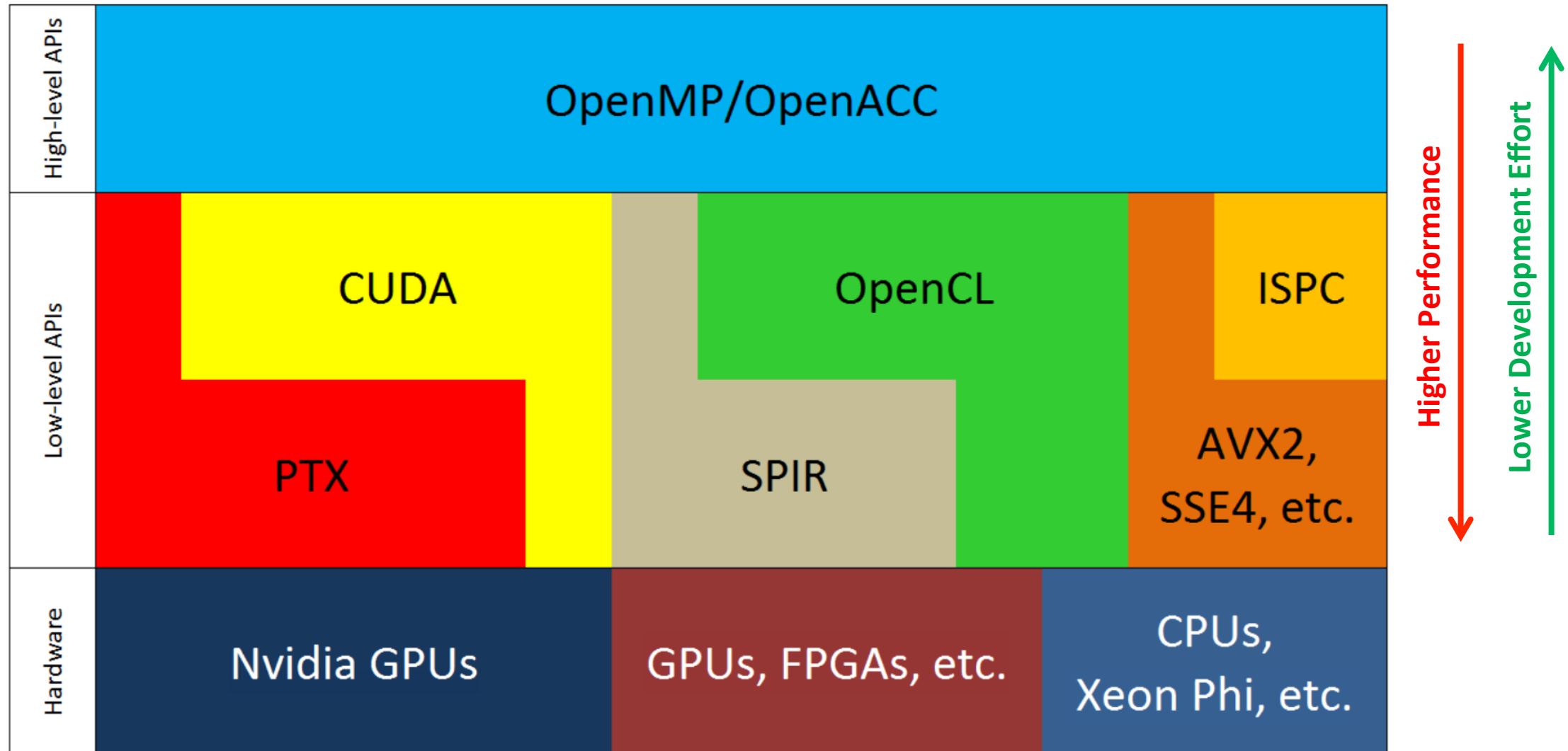
ECE Department

University of Victoria



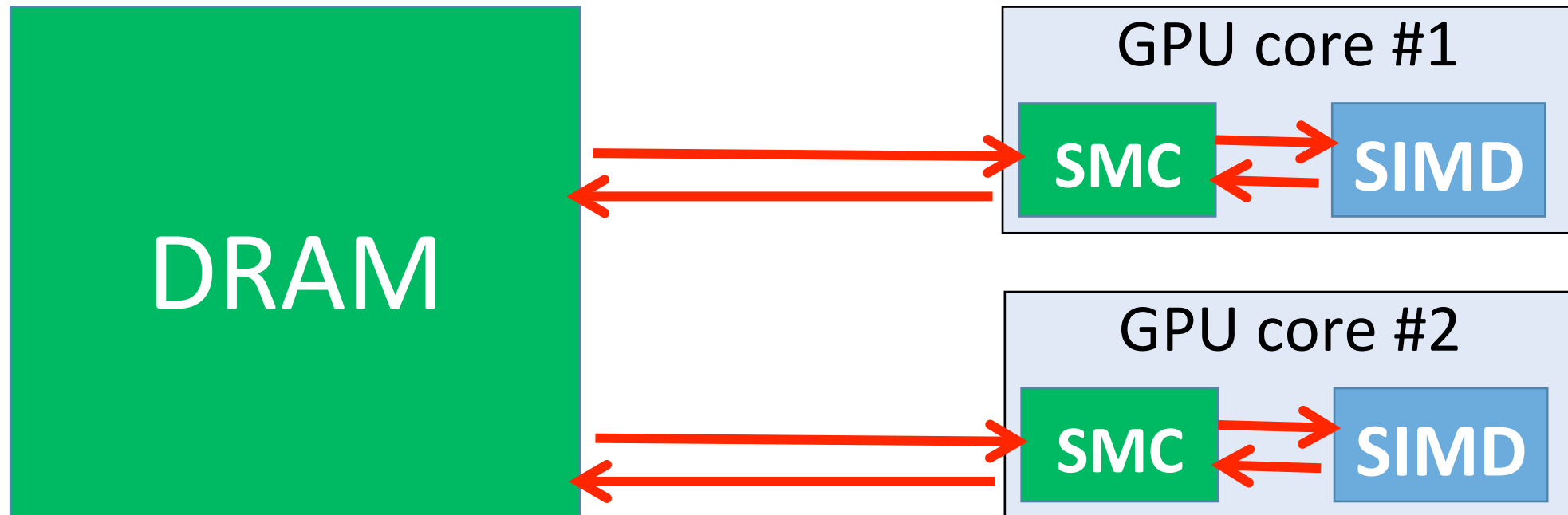
November 4, 2016

Directive-based Accelerator Programming Models



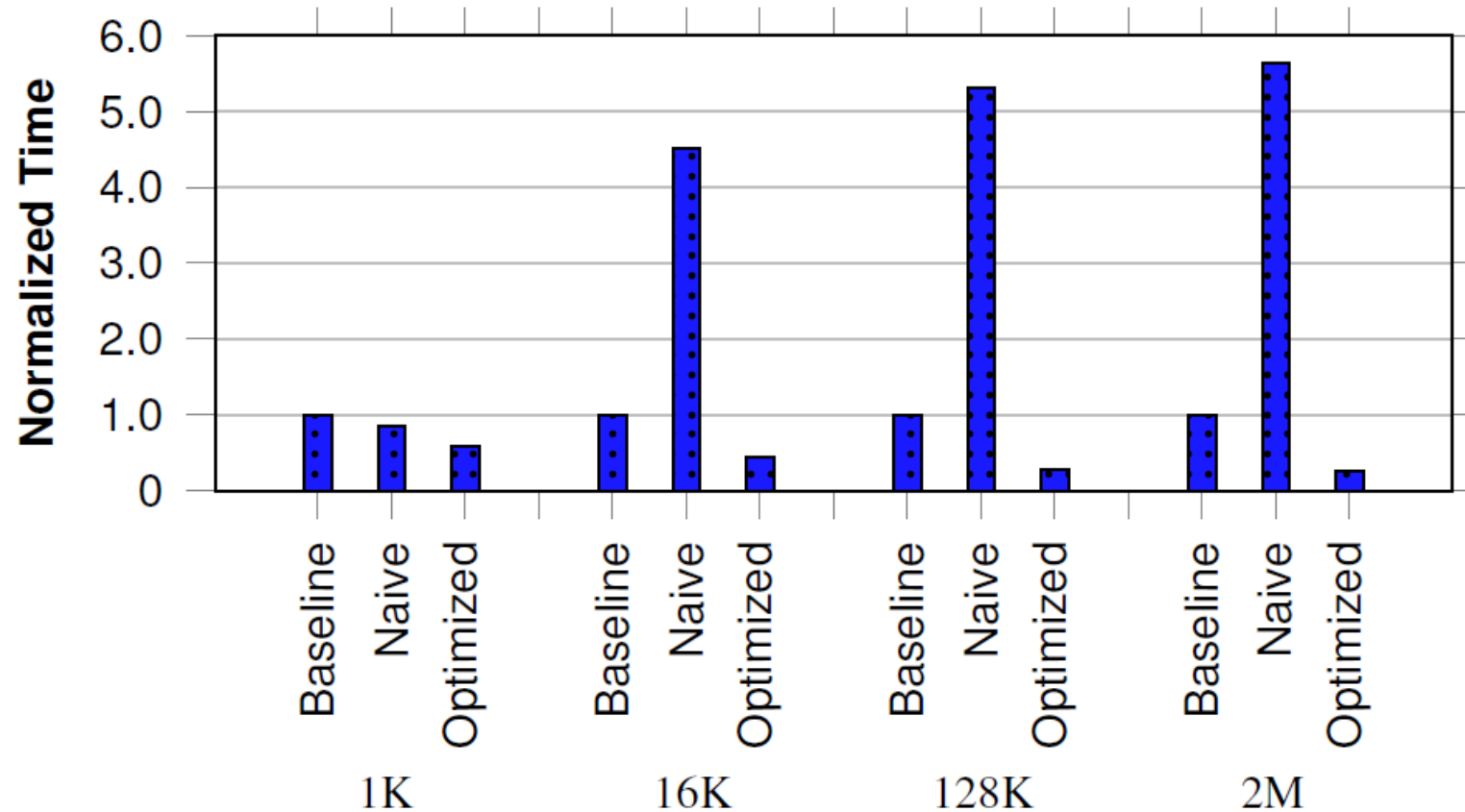
Software-managed Cache In GPUs

```
__QUALIFIER__ float smc[1024];  
smc[threadIdx.x] = 0;  
...  
float sum = a[i] + smc[threadIdx.x]  
...
```



Software-managed Cache Performance Potential In OpenACC

- 1D Stencil 30-element radius



**Naïve implementation downgrades performance, undesirably.
Cache sharing is crucial to deliver speedup over the baseline.**

Overview

- Key contribution:
 - Investigating essential optimizations to implement OpenACC cache directive efficiently
- Motivation:
 - Software-managed cache creates a significant gap between OpenACC and CUDA applications
 - OpenACC *cache* directive can narrow down this gap, potentially
 - Naïve *cache* implementation, inversely, downgrades performance
- Goals:
 - Investigating challenges in implementing *cache* directive efficiently
 - Introducing compiler passes to implement *cache* directive
- Findings:
 - Sharing cache space among chunk of parallel iterations is essential optimization
 - Best implementation performs very close to CUDA equivalent

Outline

- cache Directive Syntax/Usage
- Proposed Implementations
 - Emulating Hardware Cache
 - Range-based Conservative
 - Range-based Intelligent
- Optimizations
 - Cache fetch
 - Cache sharing
 - Cache write policy
 - Index mapping
- Evaluations

cache Directive

- From OpenACC API Standard:
 - “The cache directive may appear at the top of (inside of) a loop. It specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of the loop.”

- Syntax:

```
#pragma acc cache(var-list)
{
    // cache region
}
```

var-list e.g. array[start:length]

cache Directive Usage

- Example:

```
#pragma acc data copy(a[0:LEN],b[0:LEN])
for(n=0; n<K; ++n){
    #pragma acc parallel loop
    for(i=1; i<LEN-1; ++i){
        int lower = i-1, upper = i+1;
        float sum = 0;

        for(j=lower; j<=upper; ++j)
            sum += a[j];

        b[i] = sum/(upper-lower+1);
    }
    float *tmp=a; a=b; b=tmp;
}
```


Proposed Implementations

- **First: Emulating Hardware Cache (EHC)**
 - Data & Tag arrays are allocated in the cache
 - Tag can be direct-map, set-associative, or fully-associative
- Pros and Cons:
 - Adapts to available cache size
 - Allows fully or partially caching the subarray
 - Storing Tag array shrinks effective cache size
 - Maintaining Tag array imposes performance overhead (minimum of two cache accesses per request)

Proposed Implementations (2)

- **Second: Range-based Conservative (RBC)**

- Data array is allocated in the cache
- Two pointers keep track of the elements stored in the cache; start and end
- If requests fall within start and end, data is read from the cache, otherwise global access is made.

- Pros and Cons:

- One cache access to read/write
- Simple mapping from global to cache space (1 subtraction)
- Scales to multi-dimensional by storing a pair per dimension
- Control-flow overhead to assure data is cached

Proposed Implementations (3)


- **Third: Range-based Intelligent (RBI)**
 - Same as RBC, except avoiding control-flow overhead for cache accesses
 - Assumes accesses fall within the cached range, relying on OpenACC 2.5 spec.
- Pros and Cons:
 - Cache read/write involves one subtraction (for mapping) and one cache load/write

Optimizations

- Cache fetch
- Cache sharing
- Cache write policy
- Index mapping

Optimization: Cache Fetch

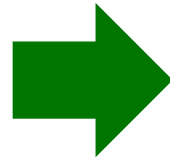
- Cache fetch routine is called before cache region starts, once per parallel iteration
 - Critical to optimize if cache region is small

```
__cache_fetch();  void __cache_fetch(subarray, start, length){  
// #pragma acc cache(subarray[])    for(int i=0; i<length; ++i)  
{                                  __cache[i] = subarray[i+start];  
    // cache region                }  
}
```

Optimization: Cache Fetch (2)

- Loop unrolling, if the subarray size is known

```
void __cache_fetch(subarray,
                  start, length){
    for(int i=0; i<length; ++i)
        __cache[i] = subarray[i+start];
}
```

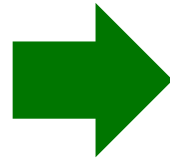


```
void __cache_fetch(subarray, start){
    // length is know to be 3, statically
    __cache[0] = subarray[0+start];
    __cache[1] = subarray[1+start];
    __cache[2] = subarray[2+start];
}
```

Optimization: Cache Fetch (3)

- Share loop among multiple subarrays

```
void __cache_fetch(subarray,
                  start, length){
    for(int i=0; i<length; ++i)
        __cache[i] = subarray[i+start];
}
```



```
void __cache_fetch(length,
                  subarr1, subarr2, subarr3,
                  start1, start2, start3){
    for(int i=0; i<length; ++i)
        __cache_s1[i] = subarr1[i+start1];
        __cache_s2[i] = subarr2[i+start2];
        __cache_s3[i] = subarr3[i+start3];
    }
}
```

Optimization: Cache Fetch (4)

- Using parallel threads to reduce loop iterations (or omitting the loop)

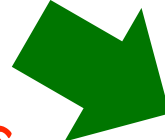
```
void __cache_fetch(subarray,  
                  start, length){  
    for(int i=0; i<length; ++i)  
        __cache[i] = subarray[i+start];  
}
```

Length unknown



```
void __cache_fetch(sa,  
                  start, length){  
    for(int i=threadIdx.x;  
          i<length;  
          i+=blockDim.x)  
        __cache[i] = sa[i+start];  
}
```

*Length is multiple of
thread block size*



```
void __cache_fetch(sa,  
                  start, length){  
    uint tid = threadIdx.x;  
    uint d   = blockDim.x;  
    __cache[tid] = sa[tid+start];  
    __cache[d*1+tid] = sa[d*1+tid+start];  
    __cache[d*2+tid] = sa[d*2+tid+start];  
    ...  
}
```


Optimization: Cache Sharing

- **What sharing means?**



```
#pragma acc parallel loop
for(i=0; i<N; i++){ // OUTER LOOP:
// depending on X and Y subarray
// main array
// iterations
#pragma acc
{
// private
// iterations
} // end
}
```

Fetch once
and share
this
private!

Iteration #1



Iteration #2



Iteration #3



Iteration #X



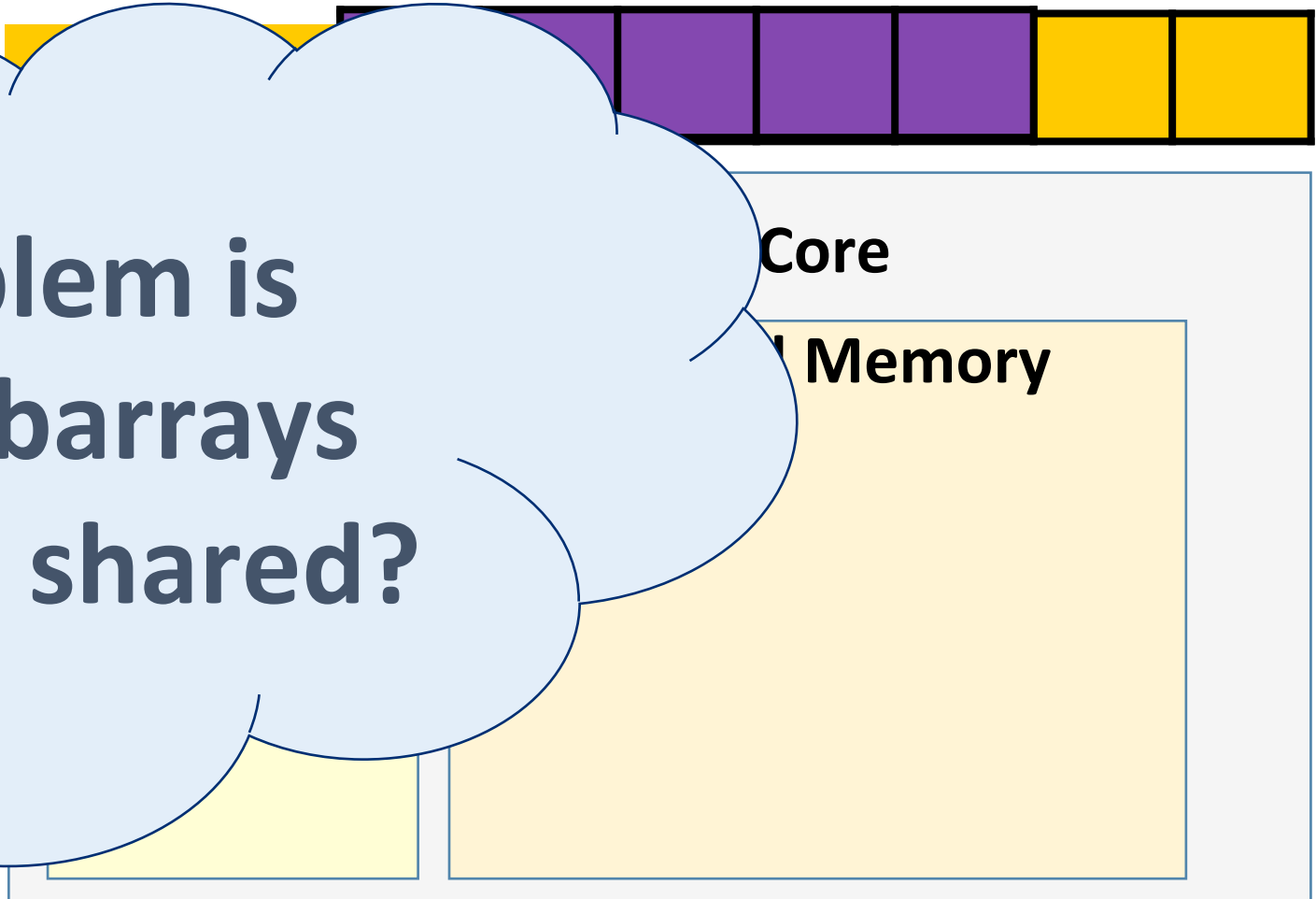
Iteration #1, #2, #3



Optimization: Cache Sharing (2)

- **How to share the cache?**

**The problem is
finding subarrays
that can be shared?**



Optimization: Cache Sharing (3)

- Problem:
 - How to find out if subarray can be shared?
- Inputs:
 - Specifications of outer parallel loops (# of iterations, increment steps)
 - Cache directive's subarray's start & length
- Output:
 - Subarrays that can be shared
 - If subarray can be shared, pointers pointing to the location of cache associated with the iteration, in the shared space
 - If subarray can be shared, size of the new shared cache

Optimization: Cache Sharing (4)

- Proposed compiler pass to find sharable caches:

input:

SBs \leftarrow subarrays listed in cache directive
IDs \leftarrow induction variables associated
with outer parallel loops
LPs \leftarrow outer parallel loops increment steps

output:

SHs \leftarrow boolean vector indicating the
subarray is shared among
iterations or not

RGs \leftarrow an expression pointing to
beginning of the range cached

for the iterations

CSs \leftarrow cache size associated with each SBs

decompose() routine calculates AST tree of start and reforms
start expression in A+B (if possible) where:

A expression is a variable listed in IDs

B expression is not composed of any variable listed in IDs

for sb in SBs:

start, length \leftarrow sb

A, B \leftarrow decompose(start, IDs)

if (A in IDs) and (abs(LP[A]) == 1):

// subarray can be shared

SHs[sb] = True

CSs[sb] = length + blockDim.x

RGs[sb] = A + B - LPs[A] * threadIdx.x

else:

SHs[sb] = False

CSs[sb] = length

RGs[sb] = start

This pass can simply be extended to support multi-dimensional subarrays.

Optimization: Cache Write Policy

- Write-back
 - Buffers cache writes and writes final changes back to DRAM at the end of the cache region
 - Keep track of dirty lines: a mask per word or assume all lines are dirty
- Write-through
 - Write data both in the cache and global memory upon every write
 - No extra work at the end of cache region

Optimization: Index Mapping

- Mapping global indexes to the locations in the cache requires mapping
 - e.g. a subtract in RBC and RBI
- For each subarray access, mapped index can be recorded in registerfile to be reused (instead of recalculating on demand)

```
...
float sum = 0;
#pragma acc cache(a[(i-1):3])
{
  for(j=lower; j<=upper; ++j)
    sum += a[j];
}
...
```

...

```
int index_0 = j-(i-1);
...
```

...

```
for(j=lower; j<=upper; ++j)
  sum += a_cache[index_0];
...
```

Evaluations Setup

- OpenACC Compiler:
 - IPMAcc + CUDA Runtime
- Hardware
 - NVIDIA Tesla K20c
- Benchmarks
 - GEMM
 - N-Body Simulation
 - Jacobi Iterations
- Code versions to compare performance and development effort:
 - Baseline OpenACC without cache directive
 - OpenACC + cache directive (with RBC or RBI implementations)
 - Hand-written CUDA optimized for shared memory

Development Effort

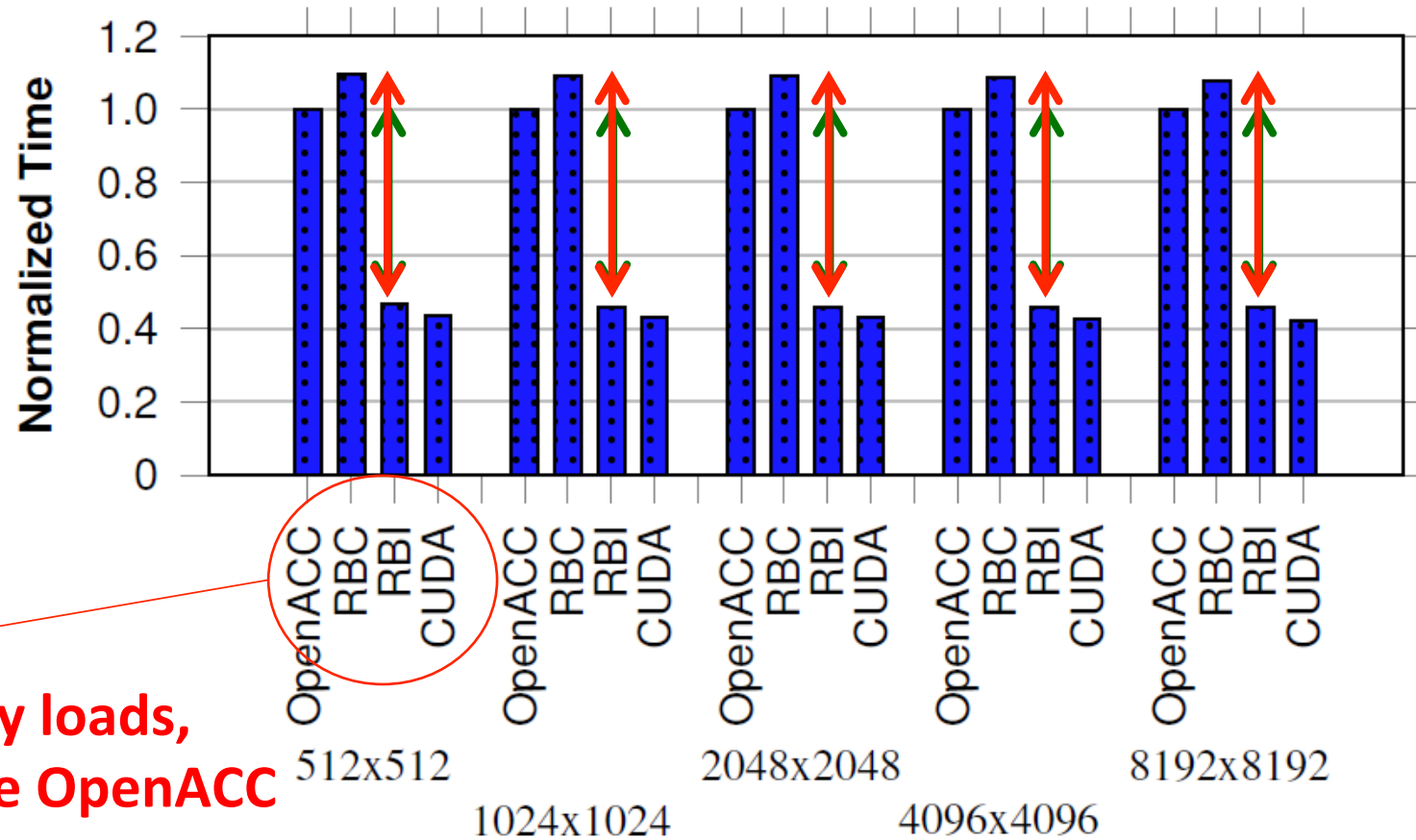
- OpenACC+cache development effort is close to OpenACC

	OpenACC	OpenACC+cache	CUDA
GEMM	84	94	116
N-Body	81	84	108
Jacobi	145	152	189

Performance

- GEMM

- RBI 2.18X faster than no-cache baseline OpenACC, 8% gap between CUDA and RBI

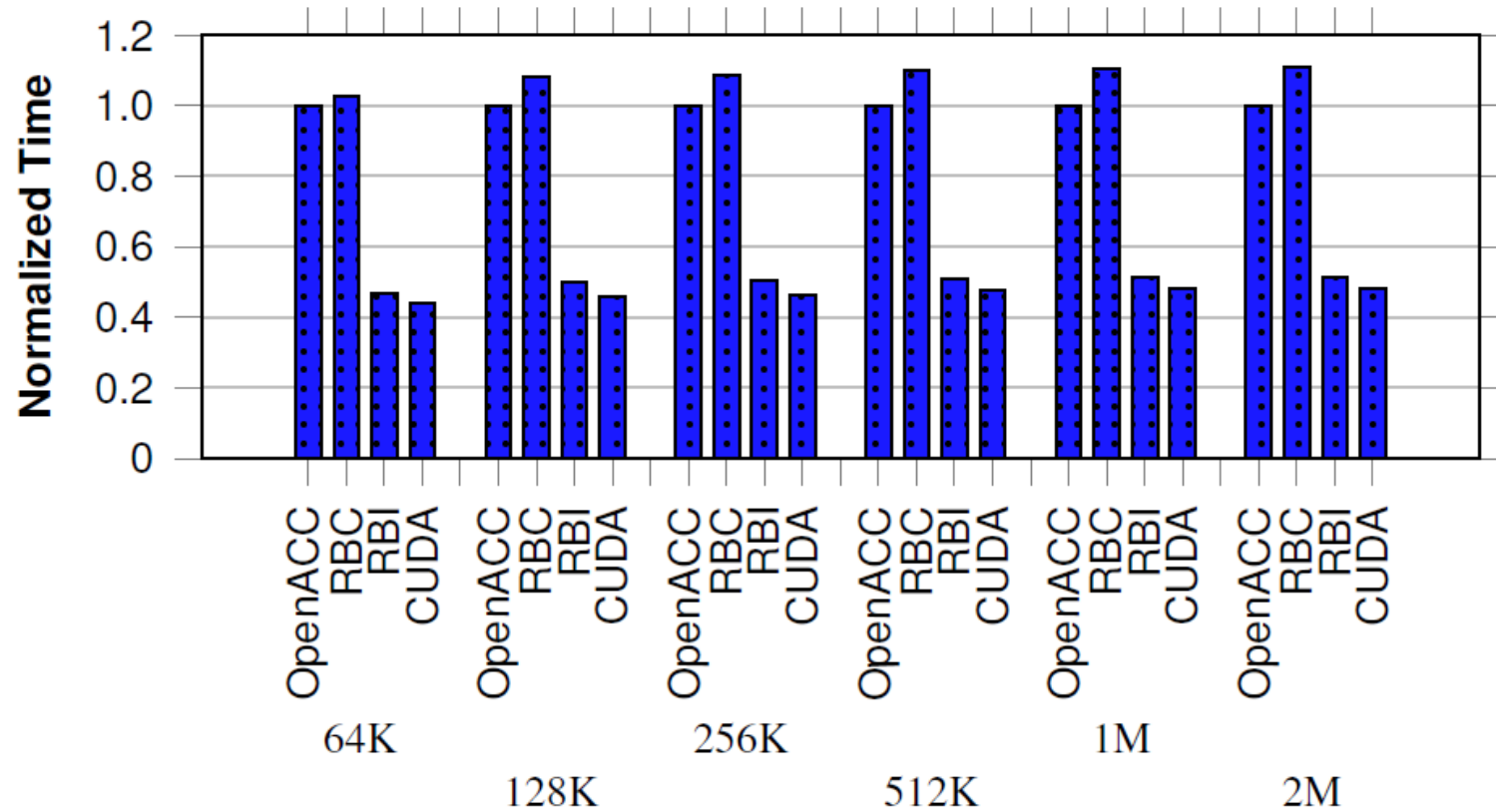


**12x saving in memory loads,
compared to baseline OpenACC**

Performance (2)

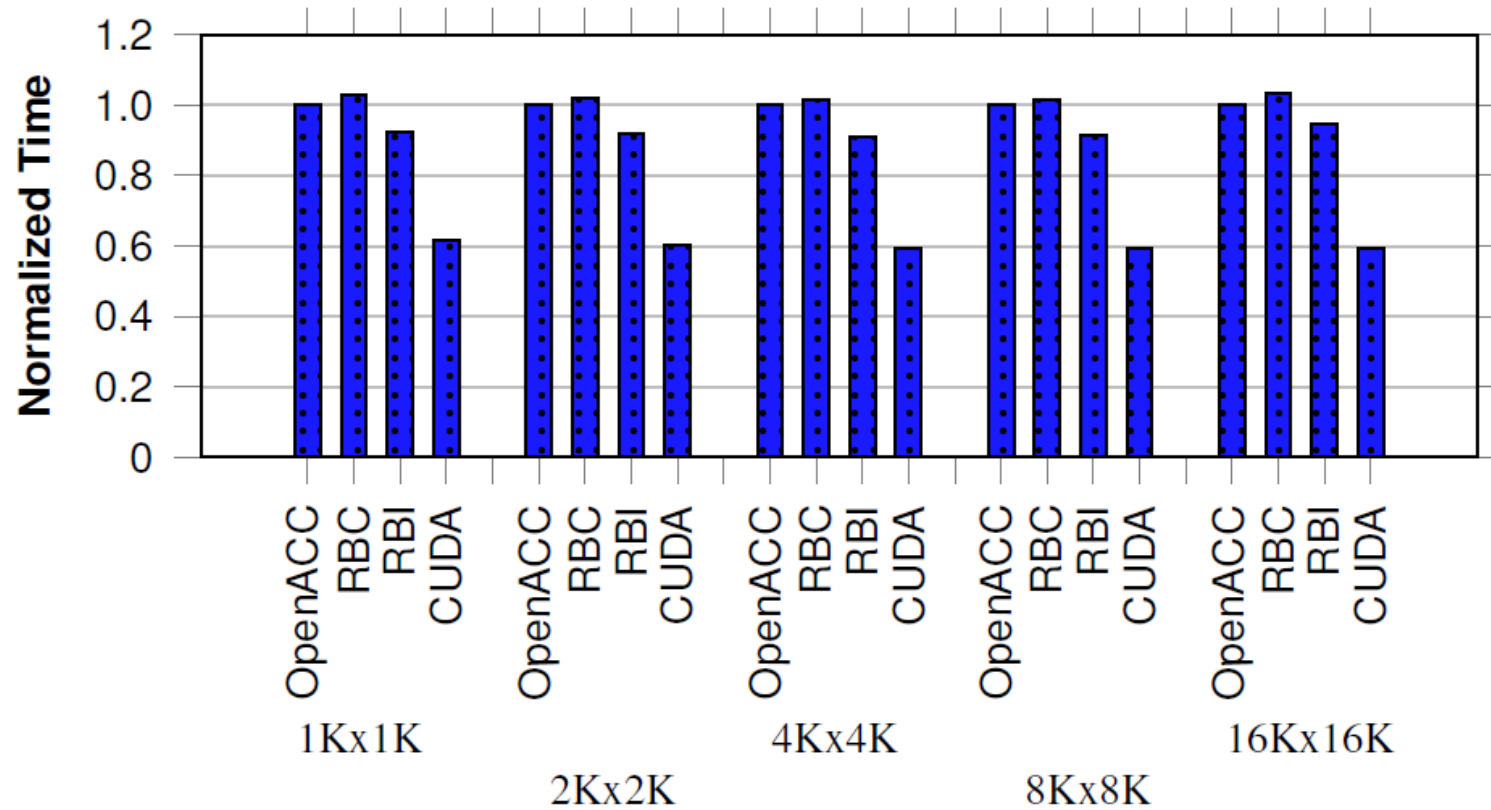
- N-Body Simulation

- RBI 95%-113% faster than no-cache baseline OpenACC, 9% gap between RBI and CUDA



Performance (3)

- Jacobi Iteration
 - RBI 6%-10% faster than no-cache baseline OpenACC,



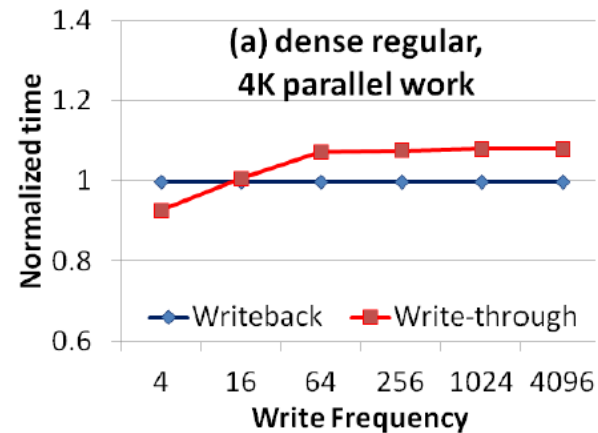
Cache Write

- Synthetic workloads with different memory patterns:
 - Dense and Regular
 - Iterations write consequent words, repeatedly
 - Sparse
 - Fraction of the iterations write arbitrary words, repeatedly
- Parameters of these workloads:
 - Parallel iterations or total work
 - Serial iterations within the work or write frequency

```
for i in 1 to total_work
#pragma acc cache(subarray[])
{
    for j in 1 to write_frequency
        write to subarray[]
}
```

Cache Write Performance Investigation

- Write-back outperform write-through when write frequency is high
- Write-through outperforms write-back when memory BW is not saturated (e.g. small dataset)



Conclusion

- Studied and addressed challenges facing cache directive implementation in CUDA
- Three different implementation proposed
- Found out cache sharing is essential to deliver performance improvement
- Found out cache directive can improve performance by 2.18x, compared to baseline OpenACC without using cache
- Implementation/Benchmarks available online:
 - <https://www.github.com/lashgar/ipmacc>
- There is more on the paper: compiler passes to support multi-dimensional array, micro-benchmarking CUDA shared memory to understand best performance.

Thank You!
Questions?

Backup Slides

EHC Example

Listing 2: Implementation of Emulating Hardware Cache (EHC) in CUDA.

```
1  __device__ void __cache_fetch(PTRTYPE* g_ptr,
2  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
3  unsigned* ctag_ptr){
4      for(unsigned i=st_idx; i<en_idx; i++){
5          unsigned cache_idx=acc_idx&0x0ff; //direct map
6          c_ptr[cache_idx]=g_ptr[i];
7          ctag_ptr[cache_idx]=i;
8      }
9  }
10 __device__ PTRTYPE __cache_read(PTRTYPE* g_ptr,
11 PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
12 unsigned acc_idx, unsigned* ctag_ptr){
13     unsigned cache_idx=acc_idx&0x0ff; //direct map
14     if(ctag_ptr[cache_idx]==acc_idx){
15         return c_ptr[cache_idx];
16     }else{
17         c_ptr[cache_idx]=g_ptr[acc_idx];
18         ctag_ptr[cache_idx]=acc_idx;
19         return c_ptr[cache_idx];
20     }
21 }
22 __device__ void __cache_write(PTRTYPE* g_ptr,
23 PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
24 unsigned acc_idx, PTRTYPE value, unsigned* ctag_ptr){
25     unsigned cache_idx=acc_idx&0x0ff; //direct map
26     if(ctag_ptr[cache_idx]!=acc_idx)
27         ctag_ptr[cache_idx]=acc_idx;
28     g_ptr[acc_idx] =c_ptr[cache_idx] =value;
29 }
```

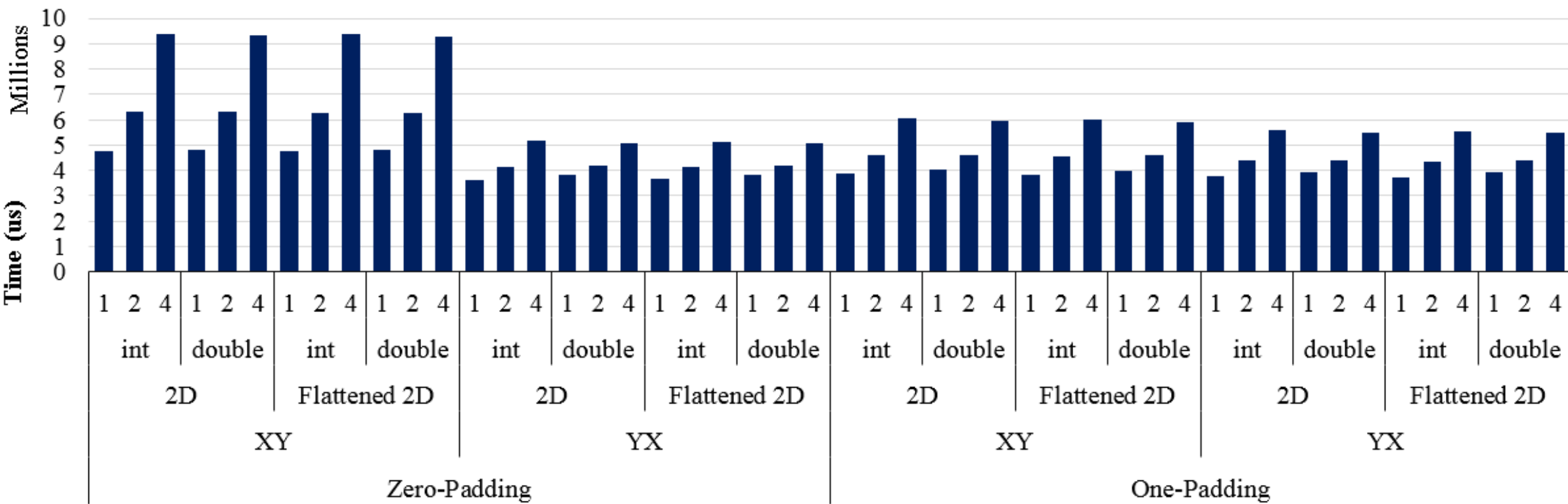
RBC Example

- RBI is the same, minus “if” control-flow statement where assumed it’s always TRUE.

Listing 3. Implementation of Range-based Conservative (RBC) in CUDA.

```
1  __device__ void __cache_fetch(PTRTYPE* g_ptr,
2  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx){
3      for(unsigned i=st_idx; i<en_idx; i++)
4          c_ptr[i-st_idx]=g_ptr[i];
5  }
6  __device__ PTRTYPE __cache_read(PTRTYPE* g_ptr,
7  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
8  unsigned acc_idx){
9      if(acc_idx>=st_idx && acc_idx<en_idx){
10         unsigned cache_idx=acc_idx-st_idx;
11         return c_ptr[cache_idx];
12     } else
13         return g_ptr[acc_idx];
14 }
15 __device__ void __cache_write(PTRTYPE* g_ptr,
16 PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
17 unsigned acc_idx, PTRTYPE value){
18     if(acc_idx>=st_idx && acc_idx<en_idx){
19         unsigned cache_idx=acc_idx-st_idx;
20         c_ptr[cache_idx]=value;
21     }
22     g_ptr[acc_idx]=value;
23 }
```

CUDA Shared Memory Bank-Conflict Evaluations



CUDA Shared Memory Bank-Conflict Evaluations (2)

```
Listing 5. CUDA microbenchmark for understanding shared memory.
// compiled for different TYPE, ITER, PAD, XY
__global__ void kernel(TYPE *GLB, int size){
    __shared__ int SHD[16+PAD][16+PAD];
    // mapping config to shared memory
    #ifdef XY
        int row=threadIdx.x, rows=blockDim.x;
        int col=threadIdx.y, cols=blockDim.y;
    #else
        int row=threadIdx.y, rows=blockDim.y;
        int col=threadIdx.x, cols=blockDim.x;
    #endif
    // fetch
    int index=(threadIdx.x+blockIdx.x*blockDim.x)*size+
              (threadIdx.y+blockIdx.y*blockDim.y);
    SHD[row][col]=GLB[index];
    // computation core
    int S = (row==(rows-1))?row:row+1;
    int N = (row==0) ?0 :row-1;
    int W = (col==(cols-1))?col:col+1;
    int E = (col==0) ?0 :col-1;
    int k=0; TYPE sum=0;
    for(k=0; k<ITER; k++){
        sum=(SHD[row][col]+ SHD[S][col]+ SHD[N][col]+
             SHD[row][E]+ SHD[row][W])*0.8;
        __syncthreads(); SHD[row][col]=sum; __syncthreads();
    }
    // write-back
    GLB[index]=SHD[row][col];
}
```